CHAPTER TEN

Implementing Programming Languages

10.1 Introduction

The Scheme system you've been using as you work through this book is itself a program, one that repeatedly reads in an expression, evaluates it, and prints out the value. The main procedure in this system is a *read-eval-print loop*. In this chapter, we'll see how such a system could have been written by building a read-eval-print loop for a somewhat stripped down version of Scheme we call Micro-Scheme.

The previous paragraph announced without any fanfare one of the deepest truths of computer science: The fully general ability to perform any computation whatsoever is itself one specific computation. The read-eval-print loop, like any other procedure, performs the one specialized task it has been programmed to do. However, its specific task is to do whatever it is told, including carrying out any other procedure. It exemplifies the *universality principle*:

> **The universality principle:** There exist universal procedures (such as the read-eval-print loop) that can perform the work of any other procedure. Like any other procedure, they are specialized, but what they specialize in is being fully general.

In the next section, we'll first describe exactly what Micro-Scheme expressions look like by using a special notation called *Extended Backus-Naur Form*. In the third section, we'll build the read-eval-print loop for Micro-Scheme. Because definitions are among the features of Scheme missing from Micro-Scheme, there is no convenient way to create recursive procedures. To overcome this, in the fourth section we'll add global definitions to Micro-Scheme, resulting in Mini-Scheme. Finally, in the application section at the end of the chapter you'll have the opportunity to modify the Mini-Scheme system so that it prints out each of the steps involved in evaluating the main problem, each subproblem, sub-subproblem, etc., much like the diagrams from the early chapters. That way, you'll have a useful tool for helping to understand Scheme evaluation.

Before launching into the development of Micro-Scheme, let's consider why we would want to build a Scheme system when we already have one available:

- As mentioned in the preceding paragraph, in the application section you'll add explanatory output that is helpful in understanding Scheme evaluation. Adding this output to the Scheme system you've been using would probably not be as easy.
- In fact, even without adding any explanatory output, you'll probably come to understand Scheme evaluation better, simply by getting an insider's perspective on it.
- You'll also be able to experiment with changes in the design of the programming language. For example, if you have been wishing that Scheme had some feature, now you'll have the opportunity to add it.
- You'll even be in a good position to implement a whole new programming language that isn't a variant of Scheme at all. Many of the general ideas of programming language implementation are independent of the specific language being implemented. The main reason why this chapter is focused on the nearly circular implementation of Mini-Scheme in Scheme is simply to avoid introducing another language for you to understand.

10.2 Syntax

The read-eval-print loop for Micro-Scheme uses many of the ideas from the movie query application in Section 7.6. There, we had a procedure, `query-loop`, that read in a query, matched it to one of a variety of patterns, took the appropriate action, and printed the result. Here, we have a loop that reads in an expression and uses a similar matching algorithm to determine what kind of expression it has. This information is then used to compute and print the value of the expression.

Recall that in the query loop, we knew that there would be some queries that didn't match any of the patterns in our database. Similarly, in the Micro-Scheme loop, there will be expressions that don't match any of the valid forms for Micro-Scheme expressions. For example, the expression `(if (not (= x 0)) (/ 2 x) (display "tried to divide by 0") 17)` is not a valid Micro-Scheme expression because there are four expressions following the symbol `if` and only three are allowed. Expressions that don't have a valid form are said to be *syntactically incorrect*;

those that are well formed are, of course, syntactically correct. Note that the emphasis is on *form*; for example, the expression `(3 2)` is syntactically correct because `2` and `3` are both valid expressions, and any collection of one or more valid expressions surrounded by parentheses is also a valid expression. However, that expression doesn't have any meaning or value. Such an error is called a *semantic error*.

The input to the movie query system was fairly easy to specify—it was a list of symbols—but the input to the Micro-Scheme read-eval-print loop has considerably more structure. Micro-Scheme is just a stripped down version of Scheme; essentially it has all the features of Scheme that we've seen up until now except `define`, `cond`, `let`, `and`, `or`, and most of the built-in Scheme procedures. This means that a Micro-Scheme expression could be a symbol, a constant (i.e., a number, boolean, or string), or a list of Micro-Scheme expressions and keywords. The *keywords* are the special symbols `if`, `lambda`, and `quote`; we'll say more about `quote`, which you haven't previously seen, in a bit. Not everything a Micro-Scheme user types in is going to be a valid Micro-Scheme expression, so we'll call each input to the read-eval-print loop a *potential Micro-Scheme expression*, or *PMSE* for short. We can give a recursive definition of a PMSE:

PMSE: A PMSE is a symbol, a number, a string, a boolean, or a list of PMSEs.

The main task of this section is to describe which PMSEs are actually Micro-Scheme expressions. To do this, we'll use a concise notation called EBNF that is commonly used for defining the syntax of formal languages, such as programming languages. The name EBNF stands for Extended Backus-Naur Form, because this notation is an extension to a form of syntax definition that John Backus developed and Peter Naur popularized by using it in the published definition of the programming language Algol, which he edited.

EBNF is one example of a notation for language *grammars*, which specify how *syntactic categories* are recursively structured. The basic idea is to be able to say things like "any collection of one or more expressions surrounded by parentheses is also an expression," which is an inherently recursive statement. The only difference is that rather than saying it in English, we have a notation for saying it that is both more precise and more concise. Regarding precision, notice that the English version could be misread as saying that each of the individual expressions is surrounded by parentheses, rather than the whole collection. Regarding concision, here is the EBNF version:

$$\langle\text{expression}\rangle \longrightarrow \texttt{(}\langle\text{expression}\rangle^{+}\texttt{)}$$

This collection of symbols with an arrow in it is called a *production* of the grammar. The arrow separates the production into two sides, the left-hand and the right-hand sides. The word ⟨expression⟩ with the angle brackets around it is a *syntactic category* *name* or *nonterminal*. A grammar is a collection of productions that is used to define one specific *syntactic category*; for Micro-Scheme it would be ⟨expression⟩. However, along the way we may want to define other syntactic categories, such as ⟨conditional⟩. The meaning of a production is that the right-hand side specifies one form that is permissible for the syntactic category listed on the left-hand side. For example, the above production gives one form that an ⟨expression⟩ can have.

The parentheses in the example production's right-hand side are necessary symbols that must appear in any ⟨expression⟩ of that form; these are called *terminal* symbols. For another example, the production

⟨expression⟩ ⟶ `(if` ⟨expression⟩ ⟨expression⟩ ⟨expression⟩`)`

contains the keyword `if` as a terminal symbol as well as the parentheses.

At this point we have two productions for ⟨expression⟩, because we have given two different forms that ⟨expression⟩s can have. This is normal; many syntactic categories will be specified by a collection of productions specifying alternative forms the category can have. The grammar is easier to read if all the productions for a particular category are grouped together; a notational shorthand is generally used for this. In the case of our two productions for ⟨expression⟩, this shorthand notation would be as follows:

⟨expression⟩ ⟶ `(`⟨expression⟩$^+$`)`
  | `(if` ⟨expression⟩ ⟨expression⟩ ⟨expression⟩`)`

The vertical bar is used to indicate that another production for the same left-hand side follows. Any number of productions can be grouped together in this way. If the right-hand sides are short, they can be listed on the same line, as follows:

⟨digit⟩ ⟶ `0` **|** `1` **|** `2` **|** `3` **|** `4` **|** `5` **|** `6` **|** `7` **|** `8` **|** `9`

Note, incidentally, that none of these productions for ⟨digit⟩ contains any nonterminal symbols on the right-hand side. Every grammar must have some productions like that to provide the base case for the recursion inherent in grammars.

The first production given for ⟨expression⟩ had a superscript plus sign in its right-hand side; this is a special notation that means "one or more." In particular, ⟨expression⟩$^+$ is the EBNF way to say "one or more ⟨expression⟩s," which was used to say that one form an ⟨expression⟩ can have is a pair of parentheses surrounding one or more ⟨expressions⟩s.

There is another very similar notation that can be used to say "zero or more." For example, suppose we want to specify the syntax of `lambda` expressions. We'll limit the body to a single ⟨expression⟩ but will allow the parameter list to have zero or more ⟨name⟩s in it so that we can have procedures with any number of parameters, including parameterless procedures. This would be expressed as follows:

$\langle$expression$\rangle$ $\longrightarrow$ `(lambda (`$\langle$name$\rangle^*$`)` $\langle$expression$\rangle$`)`

The general rule is that a syntactic category name with a superscript asterisk indicates zero or more instances of the category, whereas a syntactic category name with a superscript plus sign indicates one or more instances of the category.

Now that we have the basics of EBNF, we can use it to describe all of Micro-Scheme. Recall that Micro-Scheme is a stripped-down version of Scheme; specifically, it includes many of the features of Scheme that we've seen up until now. The basic syntactic category in Micro-Scheme is the expression.

$\langle$expression$\rangle$ $\longrightarrow$ $\langle$name$\rangle$ **|** $\langle$constant$\rangle$ **|** $\langle$conditional$\rangle$ **|** $\langle$abstraction$\rangle$
  | $\langle$application$\rangle$

$\langle$constant$\rangle$ $\longrightarrow$ $\langle$literal$\rangle$ **|** $\langle$quotation$\rangle$

$\langle$literal$\rangle$ $\longrightarrow$ $\langle$number$\rangle$ **|** $\langle$boolean$\rangle$ **|** $\langle$string$\rangle$

$\langle$conditional$\rangle$ $\longrightarrow$ `(if` $\langle$expression$\rangle$ $\langle$expression$\rangle$ $\langle$expression$\rangle$`)`

$\langle$abstraction$\rangle$ $\longrightarrow$ `(lambda (`$\langle$name$\rangle^*$`)` $\langle$expression$\rangle$`)`

$\langle$quotation$\rangle$ $\longrightarrow$ `(quote` $\langle$datum$\rangle$`)`

$\langle$application$\rangle$ $\longrightarrow$ `(`$\langle$expression$\rangle^+$`)`

$\langle$name$\rangle$ $\longrightarrow$ *any symbol allowed by the underlying Scheme except* `lambda`, `quote`, *and* `if`
$\langle$number$\rangle$ $\longrightarrow$ *any number allowed by the underlying Scheme*
$\langle$string$\rangle$ $\longrightarrow$ *any string allowed by the underlying Scheme*
$\langle$boolean$\rangle$ $\longrightarrow$ *any boolean allowed by the underlying Scheme*
$\langle$datum$\rangle$ $\longrightarrow$ *any datum allowed by the underlying Scheme*

You will notice that there are five syntactic categories at the end of the grammar that are defined in terms of the underlying Scheme. The last one, $\langle$datum$\rangle$, includes the other four as well as lists and a couple other Scheme types we have not yet discussed; specifically, $\langle$datum$\rangle$ consists of everything that Scheme will successfully read using the built-in `read` procedure. In fact, the main reason that we describe $\langle$name$\rangle$s, $\langle$number$\rangle$s, $\langle$string$\rangle$s, $\langle$boolean$\rangle$s, and $\langle$datum$\rangle$s in terms of the underlying Scheme is that we're using the built-in `read` procedure for reading in the PMSEs. Once we've read in a PMSE, the underlying Scheme has it all nicely packaged for us so we can tell if it's a symbol, a number, a boolean, a string, a list, or none of the above simply by using predicates such as `symbol?`, `number?`, and so on.

Our grammar provides two ways to specify a ⟨constant⟩. One is as a ⟨literal⟩, such as `31`, `#t`, or `"hello"`. The other way is as a ⟨quotation⟩, such as `(quote x)` or `(quote (1 2))`. In normal Scheme, you are used to seeing quotations written a different way, as `'x` or `'(1 2)`, which is really just a shorthand notation; when the `read` procedure sees `'x` in the input, it returns the list `(quote x)`.

Finally, you'll notice that we used an unfamiliar name for the syntactic category of `lambda` expressions: We called them ⟨abstraction⟩s. We didn't want to name the syntactic category ⟨lambda-expression⟩ because that would be naming it after the keyword occurring in it—naming it after what the expressions look like rather than after their meaning. (An analogy would be if we had named ⟨application⟩s "parenthesized expressions" because they have parentheses around them, rather than focusing on the fact that they represent the application of a procedure to its arguments.) We didn't want to call these expressions ⟨procedure⟩s either because a procedure is the value that results from evaluating such an expression, and we want to distinguish the expression from the value. There is a long tradition of calling this kind of expression an *abstraction*, so we adopted this name.

Exercise 10.1

The categories ⟨number⟩, ⟨string⟩, and ⟨boolean⟩ are directly testable by the corresponding Scheme procedures `number?`, `string?`, and `boolean?`, but ⟨name⟩ does not have an exact Scheme correlate. You will write one in this exercise.

a. Recall that the symbols `lambda`, `quote`, and `if` that are disallowed as names because of their special usage in Micro-Scheme are called *keywords*. Write a predicate `keyword?` that tests whether its argument is a keyword.

b. Write the predicate `name?`. You will need to use the built-in Scheme procedure `symbol?`.

Exercise 10.2

Even when a category is directly testable by Scheme, using EBNF to express it at a more primitive level can help you appreciate the expressive power of EBNF. In this exercise you will use EBNF to describe certain kinds of numbers—a small subset of those allowed by Scheme.

a. Write a production for ⟨unsigned-integer⟩. You can use the productions for ⟨digit⟩ given above.

b. Next write productions for ⟨integer⟩; an ⟨integer⟩ may start with a `-` sign, a `+` sign, or neither.

c. Finally, write productions for ⟨real-number⟩, which are (possibly) signed numbers that may have a decimal point. Note that if the real number has a decimal point, there must be at least one digit to the left or to the right (or both) of the decimal point. Thus, −43., .43, 43, +43.21, and 43.0 are all valid real numbers.

Exercise 10.3

In Section 8.3 we considered expression trees for simple arithmetic expressions. All such expressions are either numbers or lists having an operator (one of `+`, `-`, `*`, or `/`) and two operands. Actually, there are three important variants, depending on where the operator occurs: in the first position (prefix or Scheme notation), the second position (infix or standard notation), or the third position (postfix, also known as Reverse Polish notation, or RPN). Let's consider how such expressions can be specified using EBNF.

a. Write productions for ⟨arithmetic-prefix-expression⟩.

b. Write productions for ⟨arithmetic-infix-expression⟩.

c. Write productions for ⟨arithmetic-postfix-expression⟩.

d. As noted in Section 8.3, a postorder traversal of an expression tree results in a list of the nodes that is identical to the language specified by ⟨arithmetic-postfix-expression⟩, except that subexpressions are not parenthesized. Revise the productions for ⟨arithmetic-postfix-expression⟩ so that subexpressions are not parenthesized. (The overall top-level expression needn't be parenthesized either.)

Exercise 10.4

Let's consider two possible additions to our Micro-Scheme grammar involving regular Scheme expressions.

a. Write a production for `let` expressions. Remember that `let` expressions allow zero or more bindings (i.e., parenthesized name/expression pairs), and the body of the `let` contains one or more expressions. You should define a separate syntactic category for ⟨binding⟩.

b. Write productions for `cond` expressions. Remember that `cond` expressions allow one or more branches, the last of which may be an `else`, and each branch has one or more expressions following the test condition.

Exercise 10.5

Our grammar for Micro-Scheme says that an ⟨application⟩ is of the form `(`⟨expression⟩$^+$`)`. Some authors prefer to instead say that it is of the form `(`⟨expression⟩

⟨expression⟩* `)`, even though this is longer and is equivalent. Speculate why it might be preferred.

We can use the productions for ⟨expression⟩ to determine whether or not `(+ 2 3)` is a syntactically valid Micro-Scheme expression. Because it matches the production for an ⟨application⟩, it will be a valid Micro-Scheme expression if and only if `+`, `2`, and `3` are valid. Now, `+` is a symbol in Scheme and not a keyword, so it is a

The Expressiveness of EBNF

If we weren't allowed to use the superscript asterisk and plus sign in EBNF, we wouldn't lose anything in terms of the power of the notation: We could still represent all the same language constructs, just using recursion. For example, rather than

⟨application⟩ ⟶ `(`⟨expression⟩$^+$`)`

we could write

⟨application⟩ ⟶ `(`⟨expressions⟩`)`

⟨expressions⟩ ⟶ ⟨expression⟩
    | ⟨expressions⟩ ⟨expression⟩

As the above example shows, although the superscripted asterisk and plus sign don't add anything to the range of languages the EBNF notation can describe, they do contribute to keeping our grammars short and easy to understand.

Having seen what happens if we eliminate the "repetition" constructs and rely only on recursion, now let's consider the reverse. Suppose we forbid all use of recursion in EBNF but allow the superscript asterisk and plus sign. We have to be clear what it means to rule out recursion: Not only are we forbidding syntactic categories from being directly defined in terms of themselves (as ⟨expressions⟩ is in the preceding), but we are also forbidding indirect recursions, such as the definition of ⟨expression⟩ in terms of ⟨application⟩, which is itself defined in terms of ⟨expression⟩. This restriction cuts into the range of languages that is specifiable. For example, consider the language specified by the following recursive EBNF grammar:

⟨parens⟩ ⟶ `()`
    | `(`⟨parens⟩`)`

(Continued)

The Expressiveness of EBNF (Continued)

Any string of one or more left parentheses followed by the same number of right parentheses is a ⟨parens⟩. Suppose we have a nonrecursive grammar that also matches all these strings (but possibly others as well). Consider a very long string of left parentheses followed by the same number of right parentheses. If the string is long relative to the size of the nonrecursive grammar, the only way this can happen is if the asterisk or plus sign is being used at some point to match a repeated substring. The part being repeated has to contain either only left parentheses or only right parentheses because otherwise its repetition would cause a right parenthesis to come before a left parenthesis. However, if the repeated part contains only one kind of parenthesis, and if we simply repeat that part more times (which the asterisk or plus sign allows), we'll wind up with an imbalance between the number of left and right parentheses. Thus the nonrecursive grammar, if it matches all the strings that ⟨parens⟩ does, must match some other strings as well that ⟨parens⟩ doesn't; in other words, we've got a language that can be specified using a recursive grammar but not a nonrecursive one.

Even with recursion allowed, EBNF isn't the ultimate in language specification; it can't specify some very simple languages. For example, suppose we want the language to allow any number of left parentheses followed by the same number of letter `a`'s followed by the same number of right parentheses. For example, `(a)` and `((aa))` would be legal but `((a))` and `((aa)` wouldn't be. There is no way to specify this language using EBNF. Even sketching the proof of this would go beyond the scope of this book, but you'll see it in a course on formal languages and automata theory. Such courses, also sometimes called "mathematical theory of computation" or "foundations of computation," go into more details on the other issues we covered in this sidebar and cover related topics as well.

⟨name⟩ in Micro-Scheme, and thus `+` is a valid Micro-Scheme expression. Similarly, `2` and `3` are numbers, so they are Micro-Scheme ⟨constant⟩s. Thus, they too are valid Micro-Scheme expressions. Hence, the whole expression `(+ 2 3)` is also valid.

Exercise 10.6

Determine which of the following PMSEs are syntactically valid Micro-Scheme expressions and explain why.

a. `(if 3 1 5)`

b. `(lambda x (+ x 2))`

c. `(((a ((b))) c))`

d. `(lambda (lambda) 3)`

e. `(lambda () lambda)`

f. `(lambda (x) (if (> x 0) x (- x) 0))`

g. `(lambda () x)`

h. `(lambda ())`

i. `(/)`

j. `(#t #f)`

As you did the exercise above, you probably matched a PMSE against the productions for a Micro-Scheme ⟨expression⟩. Whenever you found a match, you took the various parts of the PMSE and checked to see whether they were valid as well. Note that this is a form of pattern-matching similar to what you did in Section 7.6 to determine the form of a query in the movie query system.

We can use the pattern-matching mechanism from Section 7.6 to determine whether or not a PMSE is a syntactically correct Micro-Scheme expression. In particular, we'll use the procedures `matches?` and `substitutions-in-to-match`, together with a pattern/action list appropriate for Micro-Scheme. This list will have one pattern/action pair for each kind of compound expression—⟨conditional⟩, ⟨abstraction⟩, and ⟨application⟩. The matching will determine whether or not a PMSE has the correct number of "sub-PMSEs" in the correct places, and the actions will check to see if these sub-PMSEs are valid expressions. The pattern/action list will also take care of ⟨quotation⟩s, whereas we'll have to use separate checks to determine whether or not we have one of the simplest kinds of Micro-Scheme expressions, ⟨name⟩ and ⟨literal⟩, neither of which has any sub-PMSE.

Here, then, is the code for a syntax checking predicate `syntax-ok?`, together with the pattern/action list. The procedure `all-are` is a higher-order procedure from Exercise 7.49 on page 208. It takes a predicate, such as `name?` or `syntax-ok?`, and returns a procedure that determines whether or not everything in a list satisfies the original predicate. Thus, for example, the action for the pattern starting with `lambda` includes a check that all of the parameters are really names.

```
(define syntax-ok?
  (lambda (pmse)
    (define loop  ;main procedure is on next page
      (lambda (p/a-list)
        (cond ((null? p/a-list) #f)
              ((matches? (pattern (car p/a-list)) pmse)
               (apply (action (car p/a-list))
                      (substitutions-in-to-match
                       (pattern (car p/a-list))
                       pmse)))
              (else (loop (cdr p/a-list)))))) ;end of loop
    (cond ((or (number? pmse) ;main syntax-ok? procedure
               (string? pmse)
               (boolean? pmse)) ;pmse is a literal
           #t)
          ((name? pmse) #t)
          ((list? pmse) ;try matching it against the patterns
           (loop micro-scheme-syntax-ok?-p/a-list))
          (else #f))))

(define micro-scheme-syntax-ok?-p/a-list
  (list
   (make-pattern/action '(if _ _ _)
                        (lambda (test if-true if-false)
                          (and (syntax-ok? test)
                               (syntax-ok? if-true)
                               (syntax-ok? if-false))))
   (make-pattern/action '(lambda _ _)
                        (lambda (parameters body)
                          (and (list? parameters)
                               ((all-are name?) parameters)
                               (syntax-ok? body))))
   (make-pattern/action '(quote _)
                        (lambda (datum) #t))
   (make-pattern/action '(...)   ; note that this *must* come last
                        (lambda (pmses)
                          ((all-are  syntax-ok?) pmses)))))
```

Let's look at what happens if we call `syntax-ok?` on a list-structured PMSE, say, `(if 3 1 5)`. This PMSE will match the first pattern in the pattern/action list because `(if 3 1 5)` is a list with four elements and the first element is the symbol `if`. The last three elements in the PMSE are the test expression, the expression to evaluate if the test expression is true, and the expression to evaluate if the test is false. The action that corresponds to this pattern is to recursively check to see if all three of these expressions are really well-formed Micro-Scheme expressions by using the procedure `syntax-ok?` and the special form `and`.

In the example above a *mutual recursion* occurs between `syntax-ok?` and the action procedures, much like with `even-part` and `odd-part` in Section 7.5. That is, `syntax-ok?` doesn't directly invoke itself to check the validity of sub-PMSEs but rather invokes an action procedure that in turn invokes `syntax-ok?` on the sub-PMSEs. Because this will in general result in more than one recursive call to `syntax-ok?` (for example, conditionals result in three recursive calls), the net result is tree recursion. Micro-Scheme expressions have a tree-like structure similar to the expression trees in Section 8.3. The tree recursion resulting from a call to `syntax-ok?` exactly parallels the tree-like structure of the given PMSE.

Exercise 10.7

Why does the mutual recursion between `syntax-ok?` and the action procedures eventually stop when we check the syntax of `(if 3 1 5)`? Why will it eventually stop on any list-structured PMSE?

Exercise 10.8

What happens if the PMSE being checked is the empty list?

Note that there are plenty of syntactically valid Micro-Scheme expressions that are nevertheless completely nonsensical: consider, for example, `(1 5)`. This expression is a syntactically valid Micro-Scheme expression (and a syntactically valid Scheme one, too), but it doesn't have a value, because the value of `1` is the number 1, not a procedure. The point is that this expression has the correct form for Micro-Scheme expressions, and form is the only thing that EBNF specifies. The big gain with EBNF is that the productions for a language translate fairly simply into a syntax checker such as `syntax-ok?`. In the next section, we'll see that the same productions can also serve as the basis for categorizing expressions and identifying their parts in preparation for evaluating them.

Finally, we make one important remark concerning the structure of the pattern/action list. Note that the first three patterns in the pattern/action list describe list-structured PMSEs that can be identified by their size and their first element. Because of the way the pattern/action list is structured, any other nonempty list is considered to be an application. When we extend Micro-Scheme by adding new productions, we will want to maintain this property by keeping the pattern for applications at the end of the pattern/action list.

10.3 Micro-Scheme

Now that we know the syntax for Micro-Scheme, we can build a read-eval-print loop for it. The Micro-Scheme read-eval-print loop itself is quite straightforward:

```
(define read-eval-print-loop
  (lambda ()
    (display ";Enter Micro-Scheme expression:")
    (newline)
    ;;(continued)
    (let ((expression (read)))
      (let ((value (evaluate (parse expression))))
        (display ";Micro-Scheme value: ")
        (write value)
        (newline)))
    (read-eval-print-loop)))
```

Each expression is read in with `read`, then *parsed* and *evaluated*, and finally the value is written back out using `write`, with some frills provided by `newline` and `display`. (The built-in procedure `write` is just like `display` except for some details such as providing double quote marks around strings. That way you can see the difference between the string `"foo"` and the symbol `foo`, unlike when they are `display`ed.)

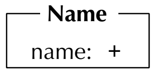
The core of this read-eval-print loop is a two-step process that uses the two procedures `parse` and `evaluate`. In order to understand the separate tasks of these two procedures, let's first consider the arithmetic expressions described in Exercise 10.3. No matter which way we denote arithmetic expressions (infix, prefix, and postfix), each expression gives rise to a unique expression tree, as described in Section 8.3. Parsing is the process of converting an actual expression to the corresponding expression tree. But why should we go through this intermediate stage (the expression tree) rather than simply evaluating the expression directly? Separating the parsing from the evaluation allows us to make changes in the superficial form or syntax of expressions (such as whether we write our arithmetic expressions in prefix, infix, or postfix) without needing to change the evaluation procedure. Furthermore, evaluation itself is made easier, because the expression tree data type can be designed for ease of evaluation rather than for ease of human writing.

Arithmetic expressions are considerably simpler than Micro-Scheme expressions in one sense, however. Namely, there were only two kinds of nodes in our expression trees: constants, which were leaves, and operators, which were internal nodes. We needed to distinguish between constants and operators in Section 8.3's `evaluate` procedure, but all internal nodes were treated the same way: by looking up and applying the specified Scheme procedure.

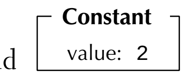
If you think instead about how Micro-Scheme works, it would be natural for expression trees to have two kinds of leaves, corresponding to the syntactic categories ⟨name⟩ and ⟨constant⟩. Each of these will need to be evaluated differently. Similarly, there are three natural candidates for kinds of internal nodes, corresponding to ⟨conditional⟩, ⟨abstraction⟩, and ⟨application⟩, because these syntactic categories have subexpressions that would correspond to subtrees. Again, the way each of these expressions is evaluated depends on what kind of expression it is. For example, think about the difference between the way `(+ (square 2) (square 3))` is evaluated and the way `(if (= x 0) 1 (/ 5 x))` is. Because we need to know what kind of expression we have in order to evaluate it, parsing must identify and mark what sort of expression it is considering and break it down into its component parts. In our example above, the expression `(+ (square 2) (square 3))` is an application, whose operator is + and whose operands are `(square 2)` and `(square 3)`. Each of these operands is itself an application with an operator, which is `square`, and an operand, which is either 2 or 3.

So, the value of `parse` will be a tree-structured data type, which is typically called an *Abstract Syntax Tree*, or *AST*. The AST for an expression indicates what kind of expression it is and what its components are. Furthermore, the components are themselves typically ASTs. The evaluation process itself can be carried out on the AST rather than the original expression; as described above, this approach has the advantage that if the language is redesigned in ways that change only the superficial syntax of expressions, only `parse` (not `evaluate`) needs to be changed.

ASTs are an abstract data type, which means we shouldn't worry too much for now about how they are represented (what they "look like") so long as they provide the appropriate operations, notably the `evaluate` operation. However, it is easier to think about ASTs if you have something concrete you can think about, so we will present here a pictorial version of ASTs that you can use when working through examples with paper and pencil. Each AST is visually represented as a tree whose root node has a label indicating what kind of AST it is. The leaf nodes, which correspond to the syntactic categories ⟨name⟩ and ⟨constant⟩, are fairly simple. For example,



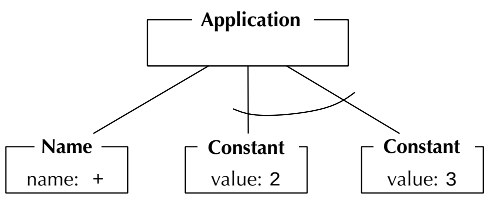
is the name AST corresponding to the name +, and



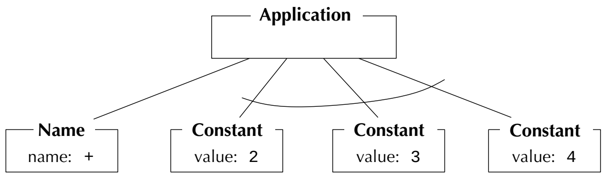
is the constant AST corresponding to 2. Note that in addition to the labels (that designate their syntactic categories Name and Constant), both of these ASTs contain information specifying which particular name or constant they represent (name: + and value: 2).

The other three syntactic categories (⟨conditional⟩, ⟨abstraction⟩, and ⟨application⟩) correspond to internal nodes because they each contain subexpressions that themselves result in ASTs. In contrast to the expression trees in Section 8.3, which always had exactly two children, the number of children of an internal node in these ASTs will vary. This number depends partially on the syntactic category; for example, the root node corresponding to the category ⟨conditional⟩ will always have three children: one each for the test, if-true, and if-false subexpressions. On the other hand, the number of children of the root node corresponding to the category ⟨application⟩ varies: The operator is one child, and the operands are the others.

First consider the ⟨application⟩ category. If we parse the Micro-Scheme expression `(+ 2 3)`, we get the following application AST:



The three children are the ASTs corresponding to the three subexpressions of the expression. The leftmost child corresponds to the operator `+`, which is a name AST, and the other children correspond to the two operands; we put a curved line in the diagram to indicate that these latter subtrees are grouped together as a *list* of operands. As noted above, the number of subtrees varies with the application; for example, parsing the expression `(+ 2 3 4)` would result in the following application AST:



We have two other kinds of ASTs: conditional ASTs, which result from parsing `if` expressions, and abstraction ASTs, which result from parsing `lambda` expressions. The conditional AST resulting from the expression `(if (= x 0) 1 (/ 5 x))` is diagrammed in Figure 10.1, and the abstraction AST resulting from the expression `(lambda (x) (* x x))` is diagrammed in Figure 10.2. Notice that the abstraction AST contains the list of parameter names and has a single sub-AST, corresponding to the body of the abstraction.

Recall that we are doing evaluation in a two-step process: first parse the expression, then evaluate the resulting AST. Thus, if we use A as a name for the first application AST shown above, the Scheme (not Micro-Scheme) expression `(parse '(+ 2 3))` has A as its value, and the Scheme expression `(evaluate` *A*`)` has 5 as its value. Those are the two steps that the Micro-Scheme read-eval-print loop goes through after reading in `(+ 2 3)`: It first parses it into the AST A, and then evaluates the AST A to get 5, which it writes back out.

What do we gain by using this two-step evaluation process? As we said at the outset, part of the gain is the decoupling of the superficial syntax (`parse`'s concern)

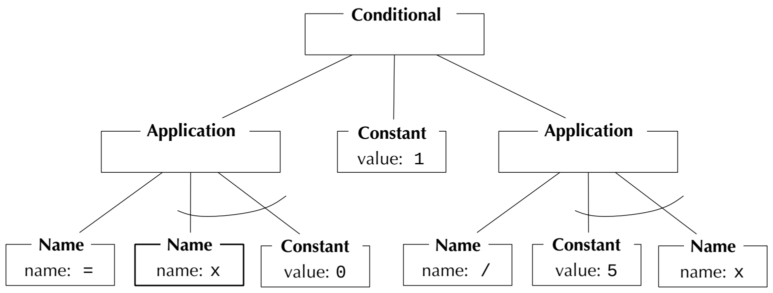


Figure 10.1 Conditional AST parsed from `(if (= x 0) 1 (/ 5 x))`

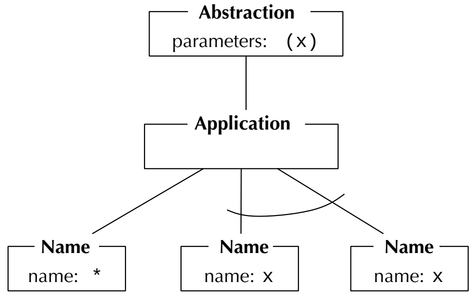


Figure 10.2 Abstraction AST parsed from `(lambda (x) (* x x))`

from the deeper structure (`evaluate`'s concern). Perhaps more important, however, is the other advantage we mentioned: The tree structure of ASTs greatly facilitates the evaluation process. ASTs are made to be evaluated. Now that we have seen AST diagrams, we can understand why this is. First, each AST has an explicit type, which controls how it is evaluated. For example, consider the two kinds of leaf nodes, name ASTs and constant ASTs. Evaluating a constant AST is trivial, because we simply return the value that it stores. Evaluating a name AST is slightly more complicated but only requires looking up its name somewhere.

As for the more complicated ASTs, their recursive structure guides the evaluation. Let's just consider how we might evaluate a conditional AST, for example, the one in Figure 10.1. In evaluating such an AST, the left child gets evaluated first, and depending on whether its value is true or false, either the second or third child is evaluated and its value is returned. The evaluation of the sub-ASTs is done recursively; how precisely a given sub-AST is evaluated depends on which kind of AST it is.

Before we start worrying about how to implement the data type of ASTs, we'll first write the procedure `parse`, assuming that we have all the constructors (`make-abstraction-ast`, `make-application-ast`, etc.) we need.

The procedure `parse` will look almost the same as the procedure `syntax-ok?` in that we need to look at the expression and see if it matches one of the forms of the expressions in our language. The only difference is that instead of returning a boolean indicating whether the syntax is okay, `parse` will return an AST. Here is the code for `parse`, together with a new pattern/action list:

```
(define parse
  (lambda (expression)
    (define loop
      (lambda (p/a-list)
        (cond ((null? p/a-list)
               (error "invalid expression" expression))
```

```
                ((matches? (pattern (car p/a-list)) expression)
                 (apply (action (car p/a-list))
                        (substitutions-in-to-match
                         (pattern (car p/a-list))
                         expression)))
                (else (loop (cdr p/a-list)))))) ;end of loop
      (cond ((name? expression) ;start of main parse procedure
             (make-name-ast expression))
            ((or (number? expression)
                 (string? expression)
                 (boolean? expression))
             (make-constant-ast expression))
            ((list? expression)
             (loop micro-scheme-parsing-p/a-list))
            (else (error "invalid expression" expression)))))

(define micro-scheme-parsing-p/a-list
  (list
   (make-pattern/action '(if _ _ _)
                        (lambda (test if-true if-false)
                          (make-conditional-ast (parse test)
                                                (parse if-true)
                                                (parse if-false))))
   (make-pattern/action '(lambda _ _)
                        (lambda (parameters body)
                          (if (and (list? parameters)
                                   ((all-are name?) parameters))
                              (make-abstraction-ast parameters
                                                    (parse body))
                              (error "invalid expression"
                                     (list 'lambda
                                           parameters body)))))
   (make-pattern/action '(quote _)
                        (lambda (value)
                          (make-constant-ast value)))
   (make-pattern/action '(...)   ; note that this *must* come last
                        (lambda (operator&operands)
                          (let ((asts (map parse
                                           operator&operands)))
                            (make-application-ast (car asts)
                                                  (cdr asts)))))))
```

Exercise 10.9

The action for ifs parses all three subexpressions into ASTs and passes the three resulting ASTs to `make-conditional-ast`. Similarly, the action for `lambda` expressions parses the body. However, it doesn't parse the parameters. Why not?

Our next task, then, is to implement the AST data structure. How are we going to do this? Although the various `make-...-ast` procedures make lots of different kinds of ASTs (one for each kind of expression), we want to be able to apply one operation to any one of them: `evaluate`. Thus, to implement ASTs we need to do so in a way that accommodates generic operations. We choose to use procedural representations, leading to the following definition of `evaluate`:

```
(define evaluate
  (lambda (ast)
    (ast 'evaluate)))
```

We'll evaluate expressions much the way we showed in Chapter 1, using the substitution model, which means that when a procedure is applied to arguments, the argument values are substituted into the procedure's body where the parameters appear, and then the result is evaluated. This process leads us to need an additional generic operator for ASTs, one that substitutes a value for a name in an AST:

```
(define substitute-for-in
  (lambda (value name ast)
    ((ast 'substitute-for) value name)))
```

Note that we've set this up so that when the AST is given the message `substitute-for`, it replies with a procedure to apply to the value and the name. That way ASTs can always expect to be given a single argument, the message (`evaluate` or `substitute-for`), even though in one case there are two more arguments to follow.

Let's look at the evaluation process and see how substitution fits into it, using our pictorial version of ASTs. We'll introduce one minor new element into our pictures, additional labels on the ASTs so that we can more easily refer to them. For example, when we talk about the AST A_2 in Figure 10.3, we mean the AST whose root node has the naming label A_2, in other words, the abstraction AST that is the full AST's first child. Suppose we parse the Micro-Scheme expression `((lambda (x) (* x x)) (+ 2 3))`, which results in the AST A_1 shown in Figure 10.3. Now let's look in detail at what happens when we evaluate A_1.

Because A_1 is an application AST, evaluating it involves first evaluating the operator AST, A_2, and the operand ASTs, of which there is only one, A_7. Because A_2 is an abstraction AST, evaluating it creates an actual procedure; let's call that procedure

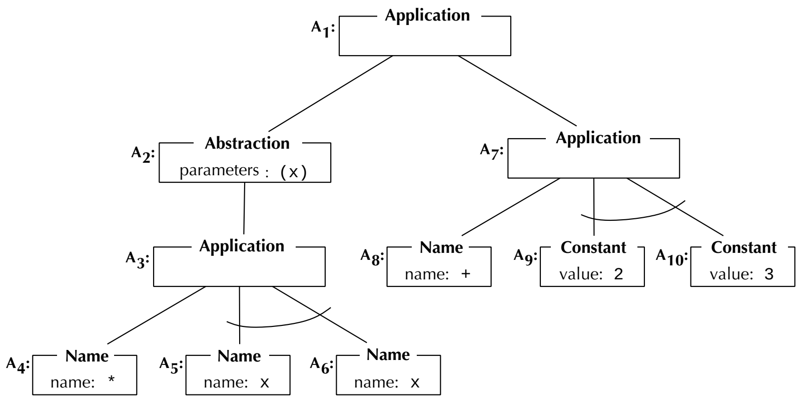


Figure 10.3 The AST corresponding to `((lambda (x) (* x x)) (+ 2 3))`.

P_1 for reference. The procedure P_1 has a parameter list that contains only `x` and has the AST A_3 as its body. Next we need to evaluate the operand, A_7, to find out what value P_1 should be applied to. Because A_7 is again an application AST, its evaluation proceeds similarly to that of A_1; we need to evaluate its operator AST, A_8, and its operand ASTs, A_9 and A_{10}. Because A_8 is a name AST, evaluating it simply involves looking up what the name `+` is a name for. The answer is that it is a name for the built-in addition procedure, which we can call P_2. Evaluating A_9 and A_{10} is even simpler, because they are constant ASTs. Each evaluates to the value shown in the AST itself: 2 for A_9 and 3 for A_{10}. Now that A_7's operator and operands have been evaluated, we can finish off evaluating A_7 by applying P_2 to 2 and 3. Doing so produces 5, because P_2 is the built-in addition procedure. Now we know that A_2's value is the procedure P_1 and that A_7's value is 5. Thus we can finish off the evaluation of A_1 by applying P_1 to 5.

Because P_1 is not a built-in procedure (unlike P_2), but rather is one that the user wrote in Micro-Scheme, we need to use the substitution model. We take P_1's body, which is the AST A_3, and replace each name AST that is an occurrence of the parameter name, `x`, by a constant AST containing the argument value, 5. We can do this task as `(substitute-for-in 5 'x` A_3`)`. The result of this substitution is the AST A_{11} shown in Figure 10.4. Notice that the AST A_4, which was the operator AST

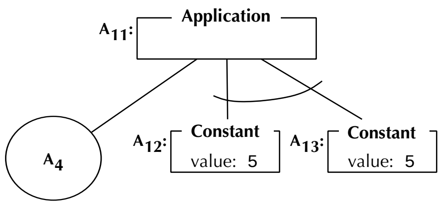


Figure 10.4 The AST resulting from `(substitute-for-in 5 'x` A_3`)`. Note that the circled A_4 indicates that an already existing AST, A_4, is being reused here.

of A_3, is also serving as the operator AST of our new A_{11}, which is what the circled A_4 indicates. Also, notice that each place where the value 5 was substituted for the name `x`, it was packaged into a constant AST; this resulted in ASTs A_{12} and A_{13}. This packaging is necessary because we can't use a naked value where a sub-AST is expected. Next we evaluate A_{11}, which involves evaluating its operator AST, A_4, and its operand ASTs, A_{12} and A_{13}. A_4 evaluates to the built-in multiplication procedure, and A_{12} and A_{13} each evaluate to 5. Finally, the built-in multiplication procedure can be applied to 5 and 5, producing the final answer of 25. This process can be shown in a diagram, as in Figure 10.5. Of course, this can also be abbreviated, for example, by leaving out the details of how substituting 5 for `x` in A_3 results in A_{11}.

We can evaluate conditional ASTs similarly to what is shown in the foregoing, but there is a bit of a twist because we first evaluate the test AST and then depending on whether its value is true or false, evaluate one of the other two sub-ASTs to provide the conditional AST's value. This process is illustrated in Figure 10.6, which shows the evaluation of an AST (A_{14}) that results from parsing `(if #f 1 2)`.

Exercise 10.10

Draw a diagram showing the AST resulting from parsing `((lambda (x) (if (> x 0) x 0)) (- 0 3))`. Now step through the process of evaluating that AST, analogously to the above evaluations of A_1 and A_{14}.

Now we're in a position to start writing the various AST constructors, each with its own method of evaluating and substituting. We start with the simplest ASTs, names and constants.

Names can be evaluated using the `look-up-value` procedure from Chapter 8; substituting a value for a name in a name AST is either a nonevent or a real substitution, depending on whether the two names are equal or not:

```
(define make-name-ast
  (lambda (name)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate) (look-up-value name))
              ((equal? message 'substitute-for)
               (lambda (value name-to-substitute-for)
                 (if (equal? name name-to-substitute-for)
                     (make-constant-ast value)
                     the-ast)))
              (else (error "unknown operation on a name AST"
                           message)))))
    the-ast))
```

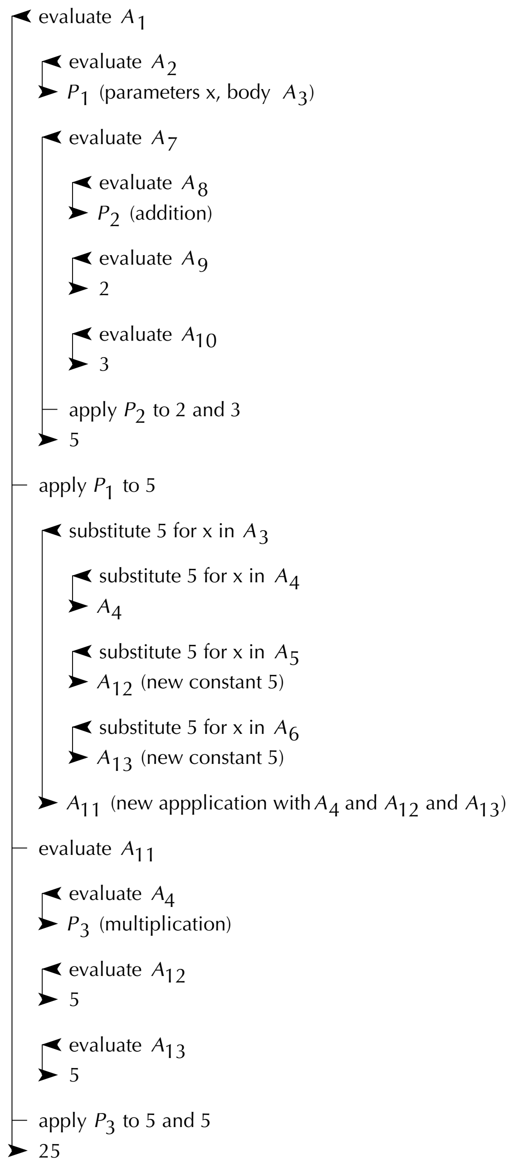


Figure 10.5 The process of evaluating the AST A_1

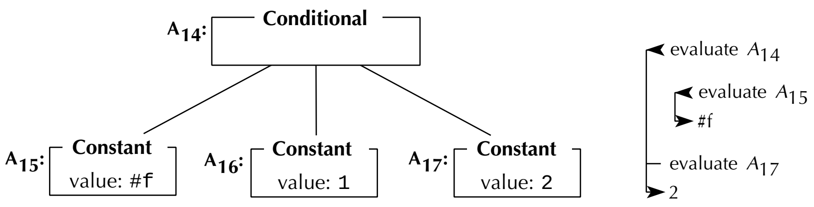


Figure 10.6 The process of evaluating a conditional AST

Exercise 10.11

Extend `look-up-value` to include all your other favorite Scheme predefined names so that they are available in Micro-Scheme as well.

Exercise 10.12

Further extend `look-up-value` so that some useful names are predefined in Micro-Scheme that *aren't* predefined in Scheme.

Constants are the ASTs that have the most straightforward implementation:

```
(define make-constant-ast
  (lambda (value)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate) value)
              ((equal? message 'substitute-for)
               (lambda (value name)
                 the-ast))
              (else (error "unknown operation on a constant AST"
                           message)))))
    the-ast))
```

The compound ASTs are much more interesting to implement, mostly because evaluating them usually involves evaluating one or more of their components. Here is the AST for conditional expressions (`if`s):

```
(define make-conditional-ast
  (lambda (test-ast if-true-ast if-false-ast)
    (lambda (message)
      (cond ((equal? message 'evaluate)
             ;;(continued)
```

```
                  (if (evaluate test-ast)
                      (evaluate if-true-ast)
                      (evaluate if-false-ast)))
                 ((equal? message 'substitute-for)
                  (lambda (value name)
                    (make-conditional-ast
                     (substitute-for-in value name test-ast)
                     (substitute-for-in value name if-true-ast)
                     (substitute-for-in value name if-false-ast))))
                 (else (error "unknown operation on a conditional AST"
                              message))))))
```

This code follows a very simple pattern: Evaluating the conditional AST involves evaluating component ASTs (first the test and then one of the others based on the result of that first evaluation), and similarly, substituting into the AST involves substituting into the constituent AST components.

Evaluating an application is similar to evaluating a conditional. First, we need to evaluate the operator and each of the operands. Then we should apply the operator's value to the values of the operands, using the built-in procedure `apply`, which assumes that an operator's value is actually a Scheme procedure. Doing a substitution on an application involves substituting into the operator and each of the operands. Therefore, in Scheme, we have

```
(define make-application-ast
  (lambda (operator-ast operand-asts)
    (lambda (message)
      (cond ((equal? message 'evaluate)
             (let ((procedure (evaluate operator-ast))
                   (arguments (map evaluate operand-asts)))
               (apply procedure arguments)))
            ((equal? message 'substitute-for)
             (lambda (value name)
               (make-application-ast
                (substitute-for-in value name operator-ast)
                (map (lambda (operand-ast)
                       (substitute-for-in value name operand-ast))
                     operand-asts))))
            (else (error "unknown operation on an application AST"
                         message))))))
```

The most complicated ASTs are probably those for abstractions (`lambda` expressions). As we mentioned previously, the result of evaluating an abstraction AST should be an actual Scheme procedure; we'll ignore that for now by assuming that we can write a procedure called `make-procedure` that will make this Scheme procedure for us. The method for handling substitutions is worth looking at closely:

```
(define make-abstraction-ast
  (lambda (parameters body-ast)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate)
               (make-procedure parameters body-ast))
              ((equal? message 'substitute-for)
               (lambda (value name)
                 (if (member name parameters)
                     the-ast
                     (make-abstraction-ast
                      parameters
                      (substitute-for-in value name body-ast)))))
              (else (error "unknown operation on an abstraction AST"
                           message)))))
    the-ast))
```

You should have noticed that if a substitution is performed where the name being substituted for is one of the parameters, the AST is returned unchanged. Only if the name isn't one of the parameters is the substitution done in the body. In other words, if we substitute 3 for `x` in `(lambda (x) (+ x y))`, we get `(lambda (x) (+ x y))` back unchanged, but if we substitute 3 for `y` in `(lambda (x) (+ x y))`, we get `(lambda (x) (+ x 3))`. This rule is what is called only substituting for *free* occurrences of the name rather than also *bound* occurrences. This limited form of substitution is the right thing to do because when we are evaluating an expression like

```
((lambda (x)
   (+ x
      ((lambda (x) (* x x))
       5)))
 3)
```

we want to substitute the 3 only for the outer `x`, not the inner one, which will later have 5 substituted for it. That way we get 28 rather than 12.

Exercise 10.13

Draw the pictorial form of the AST that would result from parsing the above expression, and carefully step through its evaluation, showing how the value of 28 is arrived at. As additional checks on your work, the parsing step should result in 13 ASTs (the main AST with 12 descendant ASTs below it), and six more ASTs should be created in the course of the evaluation so that if you sequentially number the ASTs, the last one will be numbered 19. Be sure you have enough space to work in; it is also helpful to do this exercise with a partner so that you can catch each other's slips because it requires so much attention to detail.

All that is left at this point to have a working Micro-Scheme system is the `make-procedure` procedure:

```
(define make-procedure
  (lambda (parameters body-ast)
    (lambda arguments
      (define loop
        (lambda (parameters arguments body-ast)
          (cond ((null? parameters)
                 (if (null? arguments)
                     (evaluate body-ast)
                     (error "too many arguments")))
                ((null? arguments)
                 (error "too few arguments"))
                (else
                 (loop (cdr parameters) (cdr arguments)
                       (substitute-for-in (car arguments)
                                          (car parameters)
                                          body-ast))))))
      (loop parameters arguments body-ast))))
```

One minor new feature of Scheme is shown off in the above procedure, where it has `(lambda arguments ...)` instead of the usual `(lambda (...) ...)`. This expression makes a procedure that will accept any number of arguments; they get packaged together into a list, and that list is called `arguments`.

Exercise 10.14

Suppose we define (in Scheme, not Micro-Scheme) the procedure `foo` as follows:

```
(define foo (lambda x x))
```

What predefined Scheme procedure behaves exactly like `foo`?

Now that we have a working Micro-Scheme system, we can extend it either in ways that make it more similar to Scheme or in ways that make it less similar.

Exercise 10.15

Add `let` expressions to Micro-Scheme, like those in Scheme.

Exercise 10.16

Add a `with` expression to Micro-Scheme that can be used like this:

```
;Enter Micro-Scheme expression:
(with x = (+ 2 1) compute (* x x))
;Micro-Scheme value: 9
```

The meaning is the same as `(let ((x (+ 2 1))) (* x x))` in Scheme; unlike `let`, only a single name and a single body expression are allowed.

Exercise 10.17

Add some other Scheme feature of your choice to Micro-Scheme.

Exercise 10.18

Add some other non-Scheme feature of your choice to Micro-Scheme.

10.4 Global Definitions: Turning Micro-Scheme into Mini-Scheme

Using the Micro-Scheme language you can make procedures and apply them to arguments. For example, we can make a squaring procedure and apply it to 3 as follows:

```
((lambda (x) (* x x))
 3)
```

You can also give names to procedures, which will be easiest if you've added `let` expressions to Micro-Scheme, as in Exercise 10.15. In that case, you can write

```
(let ((square (lambda (x) (* x x))))
  (square 3))
```

You can even build up a succession of procedures, where later procedures make use of earlier ones. For example,

```
(let ((square (lambda (x) (* x x))))
  (let ((cylinder-volume (lambda (radius height)
                           (* (* 3.1415927 (square radius))
                              height))))
    (cylinder-volume 5 4)))
```

However, all is not well. With the language as it stands, there is no easy way to write recursive procedures (i.e., procedures that use themselves), which is a major problem, considering all the use we've been making of recursive procedures.

To resolve this problem, we'll add definitions to our language so that we can say things like

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* (factorial (- n 1))
           n))))
```

To keep matters simple, we'll stick with *global* or *top-level* definitions that are given directly to the read-eval-print loop. We won't add internal definitions nested inside other procedures. Even with only global definitions, our language suddenly becomes much more practical, so we'll rename it Mini-Scheme to distinguish it from the nearly useless Micro-Scheme.

To support global definitions and recursive procedures, we need to introduce the notion of a *global environment*. A global environment is a collection of name/value associations that reflects the global definitions that have been entered up to some point. The read-eval-print loop starts out with an initial global environment that contains the predefined names, such as `+`. Every time the read-eval-print loop is given a new global definition, a new global environment is formed that reflects that new definition as well as all prior ones. When the read-eval-print loop is given an expression, it is evaluated *in the current global environment* rather than simply being evaluated. We need to talk about evaluating in a global environment, rather than just evaluating, because evaluating `(factorial 5)` is quite different after `factorial` has been defined than it is before. Here is the Mini-Scheme read-eval-print loop that reflects these considerations:

```
(define read-eval-print-loop
  (lambda ()
    (define loop
      (lambda (global-environment)
        (display ";Enter Mini-Scheme expr. or definition:")
        (newline)
        (let ((expression-or-definition (read)))
          (if (definition? expression-or-definition)
              (let ((name (definition-name
                            expression-or-definition))
                    (value (evaluate-in
                            (parse (definition-expression
                                     expression-or-definition))
                            global-environment)))
                (display ";Mini-scheme defined: ")
                (write name)
                (newline)
                (loop (extend-global-environment-with-naming
                       global-environment
                       name value)))
              (let ((value (evaluate-in
                            (parse expression-or-definition)
                            global-environment)))
                (display ";Mini-scheme value: ")
                (write value)
                (newline)
                (loop global-environment))))))
    (loop (make-initial-global-environment))))
```

This new read-eval-print loop distinguishes between definitions and expressions using the predicate `definition?` and selects out the two components of a definition using `definition-name` and `definition-expression`. Before we move onto the more meaty issues surrounding global environments, here are simple definitions of these more superficial procedures:

```
(define definition?
  (lambda (x)
    (and (list? x)
         (matches? '(define _ _) x))))

(define definition-name cadr)

(define definition-expression caddr)
```

Returning to global environments, we now have a good start on considering them operationally, as an abstract data type. We have seen that we need two constructors, `make-initial-global-environment` and `extend-global-environment-with-naming`. The former produces a global environment that contains the predefined names, and the latter makes a new global environment that is the same as a preexisting global environment except for one new name/value association. What about selectors? We'll need a `look-up-value-in` selector, which when given a name and a global environment finds the value associated with that name in that global environment.

To see how this selector winds up getting used, we need to consider `evaluate-in`, which is the Mini-Scheme analog of Micro-Scheme's `evaluate`:

```
(define evaluate-in
  (lambda (ast global-environment)
    ((ast 'evaluate-in) global-environment)))
```

As before, the actual knowledge regarding how to evaluate is localized within each kind of AST. The only difference is that now an `evaluate-in` operation, rather than `evaluate`, is provided by each kind of AST. This new operation is applied to the global environment in which the evaluation is to occur.

When we look at name ASTs, we see the key difference between the Mini-Scheme `evaluate-in` operation, which looks up the name in the specified global environment, and the old Micro-Scheme `evaluate`:

```
(define make-name-ast
  (lambda (name)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate-in)
               (lambda (global-environment)
                 (look-up-value-in name global-environment)))
              ((equal? message 'substitute-for)
               (lambda (value name-to-substitute-for)
                 (if (equal? name name-to-substitute-for)
                     (make-constant-ast value)
                     the-ast)))
              (else (error "unknown operation on a name AST"
                           message)))))
    the-ast))
```

Constant ASTs can be implemented in a way that is very similar to Micro-Scheme, because the global environment is completely irrelevant to their evaluation:

```
(define make-constant-ast
  (lambda (value)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate-in)
               (lambda (global-environment)
                 value))
              ((equal? message 'substitute-for)
               (lambda (value name)
                 the-ast))
              (else (error "unknown operation on a constant AST"
                           message)))))
    the-ast))
```

For conditional ASTs (i.e., `if` expressions), the global environment information is simply passed down to the evaluations of subexpression ASTs:

```
(define make-conditional-ast
  (lambda (test-ast if-true-ast if-false-ast)
    (lambda (message)
      (cond ((equal? message 'evaluate-in)
             (lambda (global-environment)
               (if (evaluate-in test-ast global-environment)
                   (evaluate-in if-true-ast global-environment)
                   (evaluate-in if-false-ast global-environment))))
            ((equal? message 'substitute-for)
             (lambda (value name)
               (make-conditional-ast
                (substitute-for-in value name test-ast)
                (substitute-for-in value name if-true-ast)
                (substitute-for-in value name if-false-ast))))
            (else (error "unknown operation on a conditional AST"
                         message))))))
```

As with the constant ASTs, the global environment is irrelevant to the evaluation of abstraction ASTs (i.e., lambda expressions):

```
(define make-abstraction-ast
  (lambda (parameters body-ast)
    (define the-ast
      (lambda (message)
        ;;(continued)
```

```
          (cond ((equal? message 'evaluate-in)
                 (lambda (global-environment)
                   (make-procedure parameters body-ast)))
                ((equal? message 'substitute-for)
                 (lambda (value name)
                   (if (member name parameters)
                       the-ast
                       (make-abstraction-ast
                        parameters
                        (substitute-for-in value name body-ast)))))
                (else (error "unknown operation on an abstraction AST"
                             message)))))
      the-ast))
```

The last AST to consider is the application AST. When a procedure is applied, its body is evaluated (with appropriate parameter substitutions done) in the current global environment. Thus, we need to keep track of that global environment. In order to do this, we'll pass in the global environment as an extra argument to the Mini-Scheme procedure, before the real ones:

```
(define make-application-ast
  (lambda (operator-ast operand-asts)
    (lambda (message)
      (cond ((equal? message 'evaluate-in)
             (lambda (global-environment)
               (let ((procedure (evaluate-in operator-ast
                                             global-environment))
                     (arguments (map (lambda (ast)
                                       (evaluate-in
                                        ast
                                        global-environment))
                                     operand-asts)))
                 (apply procedure
                        (cons global-environment arguments)))))
            ((equal? message 'substitute-for)
             (lambda (value name)
               (make-application-ast
                (substitute-for-in value name operator-ast)
                ;;(continued)
                (map (lambda (operand-ast)
                       (substitute-for-in value
                                          name
                                          operand-ast))
                     operand-asts))))
            (else (error "unknown operation on an application AST"
                         message))))))
```

Of course, we'll have to change `make-procedure` so that it expects this extra first argument and uses it appropriately:

```
(define make-procedure
  (lambda (parameters body-ast)
    (lambda global-environment&arguments
      (let ((global-environment (car global-environment&arguments))
            (arguments (cdr global-environment&arguments)))
        (define loop
          (lambda (parameters arguments body-ast)
            (cond ((null? parameters)
                   (if (null? arguments)
                       (evaluate-in body-ast global-environment)
                       (error "too many arguments")))
                  ((null? arguments)
                   (error "too few arguments"))
                  (else
                   (loop (cdr parameters) (cdr arguments)
                         (substitute-for-in (car arguments)
                                            (car parameters)
                                            body-ast))))))
        (loop parameters arguments body-ast)))))
```

Exercise 10.19

Look up `lambda` expressions in the R^{4}RS (available from the web site for this book) and figure out how to rewrite `make-procedure` so that it has `(lambda (global-environment . arguments) ...)` where the above version has `(lambda global-environment&arguments ...)`.

Finally, we need to implement global environments. Because global environments are used to find a value when given a name, one simple implementation is to use procedures. Thus a global environment is a procedure that takes a name as its parameter and returns the corresponding value:

```
(define look-up-value-in
  (lambda (name global-environment)
    (global-environment name)))

(define make-initial-global-environment
  (lambda ()
    (lambda (name)
      return the built-in procedure called name)))

(define extend-global-environment-with-naming
  (lambda (old-environment name value)
    (lambda (n)
      (if (equal? n name)
          value
          (old-environment n)))))
```

As you can see, we still need to finish writing `make-initial-global-environment`. The procedure it produces, for converting a name (such as `+`) to a built-in procedure (such as the addition procedure), is very similar to `look-up-value`. However, there is one important difference. In Micro-Scheme, we could directly use the built-in procedures (such as addition) from normal Scheme; thus, `look-up-value` could return these procedures, such as the Scheme procedure called `+`. However, in Mini-Scheme this is no longer the case. In Mini-Scheme, the evaluation of an application AST no longer applies the procedure to just its arguments. Instead, it slips in the global environment as an extra argument before the real ones. Thus, if we were to use normal Scheme's `+` as Mini-Scheme's `+`, when we tried doing even something as simple as `(+ 2 2)`, we'd get an error message because the Scheme addition procedure would be applied to three arguments: a global environment, the number 2, and the number 2 again.

To work around this problem, we'll make a Mini-Scheme version of `+` and of all the other built-in procedures. The Mini-Scheme version will simply ignore its first argument, the global environment. We can make a Mini-Scheme version of any Scheme procedure using the following converter:

```
(define make-mini-scheme-version-of
  (lambda (procedure)
    (lambda global-environment&arguments
      (let ((global-environment (car global-environment&arguments))
            (arguments (cdr global-environment&arguments)))
        (apply procedure arguments)))))
```

For example, this procedure could be used as follows:

```
(define ms+ (make-mini-scheme-version-of +))

(ms+ (make-initial-global-environment) 2 2)
```
4

Now, we can finish writing `make-initial-global-environment`:

```
(define make-initial-global-environment
  (lambda ()
    (let ((ms+ (make-mini-scheme-version-of +))
          (ms- (make-mini-scheme-version-of -))
          ;; the rest get similarly converted in here
          )
      (lambda (name)
        (cond ((equal? name '+) ms+)
              ((equal? name '-) ms-)
              ;; the rest get similarly selected in here
              (else (error "Unrecognized name" name)))))))
```

Exercise 10.20

Flesh out `make-initial-global-environment`.

Exercise 10.21

Extend your solution of Exercise 10.19 to `make-mini-scheme-version-of`.

10.5 An Application: Adding Explanatory Output to Mini-Scheme

In this section, you'll modify the Mini-Scheme implementation so that each expression being evaluated is displayed. You'll then further modify the system so that varying indentation is used to show whether an expression is being evaluated as the main problem, a subproblem, a sub-subproblem, etc. You'll also modify the system to display the value resulting from each evaluation.

To display each expression as it is evaluated, we can modify the `evaluate-in` procedure. At first you might think something like the following would work:

```
(define evaluate-in    ; Warning: this version doesn't work
  (lambda (ast global-environment)
    (display ";Mini-Scheme evaluating: ")
    (write ast)
    (newline)
    ((ast 'evaluate-in) global-environment)))
```

Unfortunately, this code displays the AST being evaluated, and what the user would really like to see is the corresponding expression. Therefore, we'll instead define `evaluate-in` as follows:

```
(define evaluate-in
  (lambda (ast global-environment)
    (display ";Mini-Scheme evaluating: ")
    (write (unparse ast))
    (newline)
    ((ast 'evaluate-in) global-environment)))
```

This code uses a new generic operation on ASTs, `unparse`. This operation should recreate the expression corresponding to an AST. The `unparse` procedure itself looks much like any generic operation:

```
(define unparse
  (lambda (ast)
    (ast 'unparse)))
```

Now we have to modify each AST constructor to provide the `unparse` operation. Here, for example is `make-application-ast`:

```
(define make-application-ast
  (lambda (operator-ast operand-asts)
    (lambda (message)
      (cond ((equal? message 'unparse)
             (cons (unparse operator-ast)
                   (map unparse operand-asts)))
            ((equal? message 'evaluate-in)
             unchanged)
            ((equal? message 'substitute-for)
             unchanged)
            (else (error "unknown operation on an application AST"
                         message))))))
```

Exercise 10.22

Add the `unparse` operation to each of the other AST constructors. When you add `unparse` to `make-constant-ast`, keep in mind that some constants will need to be expressed as quotations. For example, a constant with the value 3 can be unparsed into `3`, but a constant that has the symbol `x` as its value will need to be unparsed into `(quote x)`. You can look at the `parse` procedure to see what kinds of values can serve as constant expressions without being wrapped in a quotation.

Exercise 10.23

Adding the `unparse` operation has rather unfortunately destroyed the separation of concerns between `parse` and the AST types. It used to be that only `parse` needed to know what each kind of expression looked like. In fact, most of the knowledge regarding the superficial appearance of expressions was concentrated in the parsing pattern/action list. Now that same knowledge is being duplicated in the implementation of the `unparse` operation. Suggest some possible approaches to recentralizing the knowledge of expression appearance. You need only outline some options; actually implementing any good approach is likely to be somewhat challenging.

At this point, you should be able to do evaluations (even fairly complex ones, like `(factorial 5)`) and get a running stream of output from Mini-Scheme explaining what it is evaluating. However, no distinction is made between evaluations that are stages in the evolution of the main problem and those that are subproblems (or sub-subproblems or . . .), which makes the output relatively hard to understand. We can rectify this problem by replacing `evaluate-in` with `evaluate-in-at`, which takes not only an expression to evaluate and a global environment to evaluate it in, but also a *subproblem nesting level* at which to do the evaluation. The actual evaluation is no different at one level than at another, but the explanatory output is indented differently:

```
(define evaluate-in-at
  (lambda (ast global-environment level)
    (display ";Mini-Scheme evaluating:")
    (display-times " " level)
    (write (unparse ast))
    (newline)
    ((ast 'evaluate-in-at) global-environment level)))

(define display-times
  (lambda (output count)
    (if (= count 0)
        'done
        (begin (display output)
               (display-times output (- count 1))))))
```

The AST constructors also need to be modified to accommodate this new `evaluate-in-at` operation. Here's the new `make-application-ast`, which evaluates the operator and each operand at one subproblem nesting level deeper:

```
(define make-application-ast
  (lambda (operator-ast operand-asts)
    (lambda (message)
      (cond ((equal? message 'unparse)
             unchanged)
            ((equal? message 'evaluate-in-at)
             (lambda (global-environment level)
               (let ((procedure (evaluate-in-at operator-ast
                                                global-environment
                                                (+ level 1)))
                     (arguments (map (lambda (ast)
                                       (evaluate-in-at
                                        ast
                                        global-environment
                                        (+ level 1)))
                                     operand-asts)))
                 (apply procedure
                        (cons global-environment
                              arguments)))))
            ((equal? message 'substitute-for)
             unchanged)
            (else (error "unknown operation on an application AST"
                         message))))))
```

Exercise 10.24

Modify the other AST constructors to support the `evaluate-in-at` operation. For conditionals, the test should be evaluated one nesting level deeper than the overall conditional, but the if-true or if-false part should be evaluated at the same level as the overall conditional. (This distinction is because the value of the test is not the value of the overall conditional, so it is a subproblem, but the value of the if-true or if-false part *is* the value of the conditional, so whichever part is selected is simply a later stage in the evolution of the same problem rather than being a subproblem. This reasoning is illustrated in Figure 1.2 on page 14 and Figure 10.6.)

Exercise 10.25

Modify the `read-eval-print-loop` so that it does its evaluations at subproblem nesting level 1.

Exercise 10.26

Modify `make-procedure` so that the procedures it makes expect to receive an extra level argument after the global environment argument, before the real arguments. The procedure body (after substitutions) should then be evaluated at this level. You'll also need to change `make-application-ast` to supply this extra argument and change `make-mini-scheme-version-of` to produce procedures that expect (and ignore) this extra argument.

At this point, if you try doing some evaluations in Mini-Scheme, you'll get output like the following:

```
;Enter Mini-Scheme expr. or definition:
(+ (* 3 5) (* 6 7))
;Mini-Scheme evaluating: (+ (* 3 5) (* 6 7))
;Mini-Scheme evaluating:  +
;Mini-Scheme evaluating:  (* 3 5)
;Mini-Scheme evaluating:   *
;Mini-Scheme evaluating:   3
;Mini-Scheme evaluating:   5
;Mini-Scheme evaluating:  (* 6 7)
;Mini-Scheme evaluating:   *
;Mini-Scheme evaluating:   6
;Mini-Scheme evaluating:   7
;Mini-scheme value: 57
```

On the positive side, it is now possible to see the various subproblem nesting levels. For example, `+`, `(* 3 5)`, and `(* 6 7)` are subproblems of the main problem, and `*`, `3`, `5`, `*` (again), `6`, and `7` are sub-subproblems. On the negative side, this output is still lacking any indication of the values resulting from the various nested problems (other than the final value shown for the main problem). For example, we can't see that the two multiplications produced 15 and 42 as their values. We can arrange for the value produced by each evaluation to be displayed, indented to match the "Mini-Scheme evaluating" line:

```
(define evaluate-in-at
  (lambda (ast global-environment level)
    (display ";Mini-Scheme evaluating:")
    (display-times " " level)
    (write (unparse ast))
    (newline)
    (let ((value ((ast 'evaluate-in-at) global-environment level)))
      (display ";Mini-Scheme value     :")
      (display-times " " level)
      (write value)
      (newline)
      value)))
```

With this change, we can see the values of the two multiplication subproblems as well as the addition problem. However, as you can see below, the result is such a muddled mess as to make it questionable whether we've made progress:

```
;Enter Mini-Scheme expr. or definition:
(+ (* 3 5) (* 6 7))
;Mini-Scheme evaluating: (+ (* 3 5) (* 6 7))
;Mini-Scheme evaluating:  +
;Mini-Scheme value     :  #<procedure>
;Mini-Scheme evaluating:  (* 3 5)
;Mini-Scheme evaluating:   *
;Mini-Scheme value     :   #<procedure>
;Mini-Scheme evaluating:   3
;Mini-Scheme value     :   3
;Mini-Scheme evaluating:   5
;Mini-Scheme value     :   5
;Mini-Scheme value     :  15
;Mini-Scheme evaluating:  (* 6 7)
;Mini-Scheme evaluating:   *
;Mini-Scheme value     :   #<procedure>
;Mini-Scheme evaluating:   6
;Mini-Scheme value     :   6
;Mini-Scheme evaluating:   7
;Mini-Scheme value     :   7
;Mini-Scheme value     :  42
;Mini-Scheme value     : 57
;Mini-scheme value: 57
```

This explanatory output is so impenetrable that we clearly are going to have to find a more visually comprehensible format. We'll design an idealized version of our format first, without regard to how we are going to actually produce that output. While we are at it, we can also solve another problem with our existing output: We don't currently have any way of explicitly showing that an evaluation problem is converted into another problem of the same level with the same value. Instead, the new and old problems are treated independently, and the value is shown for each (identically). For an iterative process, we'll see the same value over and over again. For example, if we computed the factorial of 5 iteratively, we'd get shown the value 120 not only as our final value but also as the value of each of the equivalent problems, such as $1 \times 5!$, $5 \times 4!$, $20 \times 3!$, etc. Yet we'd really like to see each problem converted into the next with a single answer at the bottom.

An example of our idealized format is shown in Figure 10.7; as you can see, it is closely based on the diagrams we used to explain AST evaluation. Notice that we

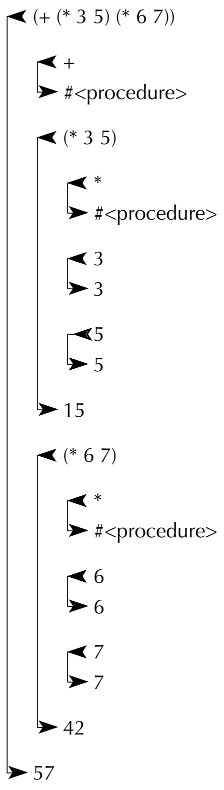
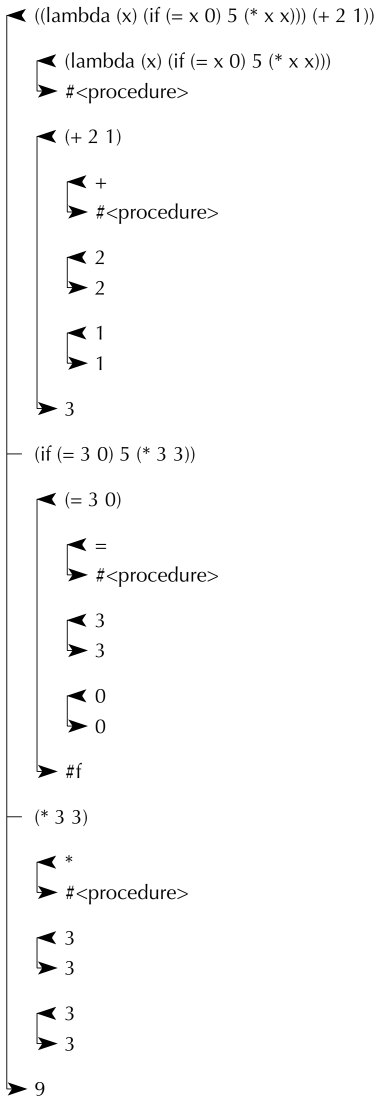


Figure 10.7 An idealized example of explanatory output

are still using indentation to show the subproblem nesting levels, but now we are also using lines with arrowheads to show the connection between each expression and its value. We can also use a similar format to show several expressions sharing the same value, as in Figure 10.8. Here three expressions all share the value 9. The first is an application expression, and the second results from it by substituting the



```
((lambda (x) (if (= x 0) 5 (* x x))) (+ 2 1))
  (lambda (x) (if (= x 0) 5 (* x x)))
  #<procedure>
  (+ 2 1)
    +
    #<procedure>
    2
    2
    1
    1
  3
(if (= 3 0) 5 (* 3 3))
  (= 3 0)
    =
    #<procedure>
    3
    3
    0
    0
  #f
(* 3 3)
  *
  #<procedure>
  3
  3
  3
  3
9
```

Figure 10.8 Another idealized example of explanatory output, with three equivalent problems sharing the value 9

argument value, 3, into the procedure body in place of the parameter name, `x`. The resulting conditional expression, `(if (= 3 0) 5 (* 3 3))`, is in turn converted into the third equivalent expression, `(* 3 3)`, because the condition evaluates to a false value.

If we want to approximate these diagrams, but do so using the normal Scheme `display` procedure, which produces textual output, we'll have to settle for using characters to approximate the lines and arrowheads. Our two examples are shown in this form in Figures 10.9 and 10.10.

```
+-< (+ (* 3 5) (* 6 7))
|
| +-< +
| +-> #<procedure>
|
| +-< (* 3 5)
| |
| | +-< *
| | +-> #<procedure>
| |
| | +-< 3
| | +-> 3
| |
| | +-< 5
| | +-> 5
| |
| +-> 15
|
| +-< (* 6 7)
| |
| | +-< *
| | +-> #<procedure>
| |
| | +-< 6
| | +-> 6
| |
| | +-< 7
| | +-> 7
| |
| +-> 42
|
+-> 57
```

Figure 10.9 Explanatory output with lines and arrowheads approximated using characters

```
+-< ((lambda (x) (if (= x 0) 5 (* x x))) (+ 2 1))
|
| +-< (lambda (x) (if (= x 0) 5 (* x x)))
| +-> #<procedure>
|
| +-< (+ 2 1)
| |
| | +-< +
| | +-> #<procedure>
| |
| | +-< 2
| | +-> 2
| |
| | +-< 1
| | +-> 1
| |
| +-> 3
|
+-- (if (= 3 0) 5 (* 3 3))
|
| +-< (= 3 0)
| |
| | +-< =
| | +-> #<procedure>
| |
| | +-< 3
| | +-> 3
| |
| | +-< 0
| | +-> 0
| |
| +-> #f
|
+-- (* 3 3)
|
| +-< *
| +-> #<procedure>
|
| +-< 3
| +-> 3
|
| +-< 3
| +-> 3
|
+-> 9
```

Figure 10.10 Second example of explanatory output using characters

In the character-based version of the explanatory output, there are two kinds of lines: lines that have something on them, like

```
| +-< (+ 2 1)
```

or

```
| +-> 3
```

or

```
+-- (* 3 3)
```

and those that are blank aside from the vertical connecting lines, such as

```
| |
```

We can use two procedures for producing these two kinds of line. For the ones that have content, we need to specify the thing to write (which might be an expression or a value), the "indicator" that shows what kind of line this is (`<` or `>` or `-`), and the nesting level. For blank lines, only the nesting level is needed:

```
(define write-with-at
  (lambda (thing indicator level)
    (display-times "| " (- level 1))
    (display "+-")
    (display indicator)
    (display " ")
    (write thing)
    (newline)))

(define blank-line-at
  (lambda (level)
    (display-times "| " level)
    (newline)))
```

Now we have to insert the appropriate calls to these procedures into our evaluator. We'll need to differentiate between two kinds of evaluations: those that should have lines with leftward pointing arrowheads (initial evaluations) and those that should have arrowheadless connecting lines (additional evaluations sharing the same ultimate value). The additional evaluations, with the arrowheadless line, originate from two sources: evaluating the body of a procedure with the argument values substituted in and evaluating one or the other alternative of a conditional. Both are shown in our example of evaluating `((lambda (x) (if (= x 0) 5 (* x x))) (+ 2 1))`. We can handle initial and additional evaluations differently by using two separate procedures. For initial evaluations we'll use our existing `evaluate-in-at`, which provides the left-arrow line and also is responsible for the right-arrow line at the end with the value. We'll use a new procedure, `evaluate-additional-in-at`, for the additional evaluations, which just "hook into" the existing evaluation's line:

```
(define evaluate-in-at
  (lambda (ast global-environment level)
    (blank-line-at (- level 1))
    (write-with-at (unparse ast) "<" level)
    (let ((value ((ast 'evaluate-in-at) global-environment level)))
      (write-with-at value ">" level)
      value)))


(define evaluate-additional-in-at
  (lambda (ast global-environment level)
    (blank-line-at level)
    (write-with-at (unparse ast) "-" level)
    ((ast 'evaluate-in-at) global-environment level)))
```

Exercise 10.27

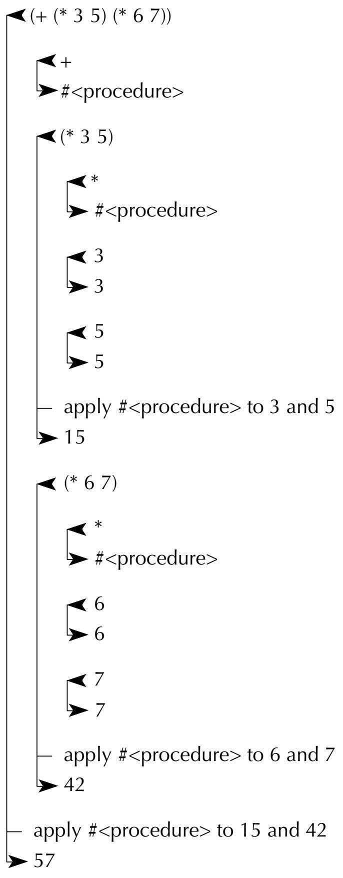
Three calls to `evaluate-in-at` need to be changed to `evaluate-additional-in-at`. Change them.

Exercise 10.28

To make the output look as shown, it is also necessary to provide a blank line before the value of a built-in procedure. Put the appropriate call to `blank-line-at` into the procedures generated by `make-mini-scheme-version-of`.

Exercise 10.29

When an application expression is evaluated, it might be desirable to explicitly show that a procedure is being applied and what argument values it is being applied to, after the operator and operands have been evaluated. Figure 10.11 shows an example of this. Add this feature.



```
(+ (* 3 5) (* 6 7))
  +
  #<procedure>
  (* 3 5)
    *
    #<procedure>
    3
    3
    5
    5
  apply #<procedure> to 3 and 5
  15
  (* 6 7)
    *
    #<procedure>
    6
    6
    7
    7
  apply #<procedure> to 6 and 7
  42
apply #<procedure> to 15 and 42
57
```

Figure 10.11 Explanatory output with applications shown

Exercise 10.30

Decide what further improvements you'd like to have in the explanatory output and make the necessary changes.

Review Problems

Exercise 10.31

Use EBNF to write a grammar for the language of all strings of one or more digits that simultaneously meet both of the following requirements:

a. The digits alternate between even and odd, starting with either.

b. The string of digits is the same backward as forward (i.e., is palindromic).

Your grammar may define more than one syntactic category name (nonterminal), but be sure to specify which one generates the language described above.

Exercise 10.32

Suppose the following Micro-Scheme expression is parsed:

```
((lambda (x) x) (if (+ 2 3) + 3))
```

a. Draw the AST that would result.

b. If this AST were evaluated, two of the ASTs it contains (as sub-ASTs or sub-sub-ASTs, etc.) would not wind up getting evaluated. Indicate these two by circling them, and explain for each of them why it doesn't get evaluated.

Exercise 10.33

In Scheme, Micro-Scheme, and Mini-Scheme, it is an error to evaluate `((+ 2 3) (* 5 7) 16)` because this will try to apply 5 to 35 and 16, and 5 isn't a procedure. It would be possible to change the language so that instead of this construction being an error, it would evaluate to the three-element list `(5 35 16)`. That is, when the "operator" subexpression of an "application" expression turns out not to evaluate to a procedure, a list of that value and the "operand" values is produced.

a. Change Micro-Scheme or Mini-Scheme to have this new feature.

b. Argue that this is an improvement to the language.

c. Argue that it makes the language worse.

Exercise 10.34

Suppose that the Micro-Scheme `make-conditional-ast` were changed to the following:

```
(define make-conditional-ast
  (lambda (test-ast if-true-ast if-false-ast)
    (lambda (message)
      (cond ((equal? message 'evaluate)
             (let ((test-value (evaluate test-ast))
                   (if-true-value (evaluate if-true-ast))
                   (if-false-value (evaluate if-false-ast)))
               ;;(continued)
```

```
          (if test-value
              if-true-value
              if-false-value)))
       ((equal? message 'substitute-for)
        (lambda (value name)
          (make-conditional-ast
           (substitute-for-in value name test-ast)
           (substitute-for-in value name if-true-ast)
           (substitute-for-in value name if-false-ast))))
       (else (error "unknown operation on a conditional AST"
                    message))))))
```

a. Give an example of a conditional expression where this new version of `make-conditional-ast` would produce an AST that evaluates to the same value as the old version would.

b. Give an example of a conditional expression where evaluating the AST constructed by the new version would produce different results from evaluating the AST produced by the old version.

c. Is this change a good idea or a bad one? Explain.

▷ Exercise 10.35

Rewrite `look-up-value` to use a table of names and their corresponding values, rather than a large `cond`.

▷ Exercise 10.36

Replace the global-environment ADT implementation with an alternative representation based on a list of name/value pairs.

▷ Exercise 10.37

Some programming languages have a so-called `arithmetic-if` expression that is similar to Scheme's `if` expression, except that instead of having a boolean test condition and two other subexpressions (the if-true and if-false subexpressions), it has a numerical test expression and *three* other subexpressions (the if-negative, the if-zero, and the if-positive subexpressions). To evaluate an `arithmetic-if` expression, you first evaluate the test expression, and then, depending upon whether that value is negative, zero, or positive, the corresponding subexpression is evaluated. For example, if you wanted to define an `expt` procedure that appropriately dealt with both positive and negative integers, you could write

```
(define expt
  (lambda (b n)
    (arithmetic-if n
                   (/ 1 (expt b (- n)))
                   1
                   (* b (expt b (- n 1))))))
```

You will work through the details of adding `arithmetic-if`'s to Mini-Scheme in this problem. To get you started, let's choose to implement `arithmetic-ifs` using a new AST constructor `make-arithmetic-if-ast`. The skeleton for `make-arithmetic-if-ast`, with the important code left out, is as follows (note that all subexpressions are passed in parsed):

```
(define make-arithmetic-if-ast
  (lambda (test-value-ast if-neg-ast if-zero-ast if-pos-ast)
    (lambda (message)
      (cond ((equal? message 'evaluate-in)
             (lambda (global-environment)

                      code for evaluate-in               ))

            ((equal? message 'substitute-for)
             (lambda (value name)

                    code for substitute-for              ))

            (else (error "unknown operation on a conditional AST"
                         message))))))
```

a. Add the code for `evaluate-in`.

b. Add the code for `substitute-for`.

c. Add the appropriate pattern/action to the `micro-scheme-parsing-p/a-list`.

▷ Exercise 10.38

Suppose we add a new kind of expression to the Micro-Scheme language, the `uncons` expression. The EBNF for it is as follows:

```
(uncons ⟨expression⟩ into ⟨name⟩ and ⟨name⟩ in ⟨expression⟩)
```

This kind of expression is evaluated as follows:

- The first ⟨expression⟩ is evaluated. Its value must be a pair (such as `cons` produces); otherwise, an error is signaled.
- The car of that pair is substituted for the first ⟨name⟩, and the cdr for the second ⟨name⟩, in the second ⟨expression⟩.
- After these substitutions have been made, the second ⟨expression⟩ (as modified by the substitutions) is then evaluated. Its value is the value of the overall `uncons` expression.

For a simple (and stupid) example, the expression

```
(uncons (cons 3 5) into x and y in (+ x y))
```

would evaluate to 8.

Parsing an `uncons` expression involves parsing the constituent expressions, which we can call the pair-expression and the body-expression. The resulting two ASTs, which we can call the pair-ast and body-ast, get passed into the `make-uncons-ast` constructor, along with the two names, which we can call the car-name and cdr-name. Here is the outline of `make-uncons-ast`; write the two missing pieces of code.

```
(define make-uncons-ast
  (lambda (pair-ast body-ast car-name cdr-name)
    (lambda (message)
      (cond ((equal? message 'evaluate)
             ┌──────────────────────────────────────┐
             │                                      │
             │          code for evaluate           │)
             │                                      │
             └──────────────────────────────────────┘
            ((equal? message 'substitute-for)
             (lambda (value name)
               ┌──────────────────────────────────────┐
               │                                      │
               │       code for substitute-for        │))
               │                                      │
               └──────────────────────────────────────┘
            (else (error "unknown operation on a for AST"
                         message))))))
```

Chapter Inventory

Vocabulary

read-eval-print loop
Extended Backus-Naur Form (EBNF)
syntactic correctness
semantic error
keyword
grammar
syntactic category
production
nonterminal
terminal
abstraction
parsing
free
bound
global or top-level definition
global environment
subproblem nesting level
arithmetic if

Slogans

The universality principle

Abstract Data Types

abstract syntax tree (AST)

New Predefined Scheme Names

`symbol?`
`string?`
`boolean?`
`write`

New Scheme Syntax

`quote`
`lambda` expressions accepting variable numbers of arguments

Scheme Names Defined in This Chapter

`keyword?`
`name?`
`syntax-ok?`
`micro-scheme-syntax-ok?-p/a-list`
`read-eval-print-loop`
`parse`
`micro-scheme-parsing-p/a-list`
`evaluate`
`substitute-for-in`
`make-name-ast`
`make-constant-ast`
`make-conditional-ast`
`make-application-ast`
`make-abstraction-ast`
`make-procedure`
`definition?`
`definition-name`
`definition-expression`
`evaluate-in`
`look-up-value-in`
`make-initial-global-environment`

```
extend-global-environment-        write-with-at
   with-naming                    blank-line-at
make-mini-scheme-version-of       evaluate-additional-in-at
unparse                           make-arithmetic-if-ast
evaluate-in-at                    make-uncons-ast
display-times
```

Sidebars

The Expressiveness of EBNF

Notes

We motivated Mini-Scheme with the remark that Micro-Scheme provides no easy way to express recursive procedures. As an example of the not-so-easy ways of expressing recursion that *are* possible even in Micro-Scheme, we offer the following:

```
;; factorial-maker makes factorial when given factorial-maker
(let ((factorial-maker
       (lambda (factorial-maker)
         (lambda (n)
           (if (= n 0)
               1
               (let ((factorial
                      (factorial-maker factorial-maker)))
                 (* (factorial (- n 1))
                    n)))))))
  (let ((factorial (factorial-maker factorial-maker)))
    (factorial 52)))
```